LA-UR-22-20545

Title: Large Program Design

Author(s): Ferenbaugh, Charles Roger

Intended for: SPEED lecture series, Tue 1/25/2022

Issued: 2022-01-24

Los Alamos
NATIONAL LABORATORY

Large Program Design

SPEED Lecture Series

Charles Ferenbaugh, CCS-7

January 25, 2022

Managed by Triad National Security, LLC, for the U.S. Department of Energy's NNSA.

Myth: “Writing code is easy - anyone can do it!”

Reality: Only partly true – it depends on what kind of code you want

An analogy:

Most people could build this

Myth: “Writing code is easy - anyone can do it!”

Reality: Only partly true – it depends on what kind of code you want
An analogy:

Most people could build this

But it’s much harder to build this

Los Alamos NATIONAL LABORATORY

Myth: “Writing code is easy - anyone can do it!”

Reality: Only partly true – it depends on what kind of code you want
An analogy:

Not just a bigger doghouse

Not just a bunch of doghouses put together

Needs to be *designed*

But it’s much harder to build this

Small vs. large software projects

- It can be OK to just throw together code that is **small**, **single-purpose**, and **short-lived**
- But there's more need for good design if the code has to:

  - Live longer
  - Give highly reliable results
    - Publications, deliverables, …
  - Be easy to use
  - Get bigger/cover more physics
    - Complexity grows as N^2
  - Cover more application domains
  - Have more users
  - Have more widely-distributed users
  - Have more developers work on it
  - Run larger problems, on large clusters
  - Run on new architectures (Cell, Xeon Phi, GPU, …)

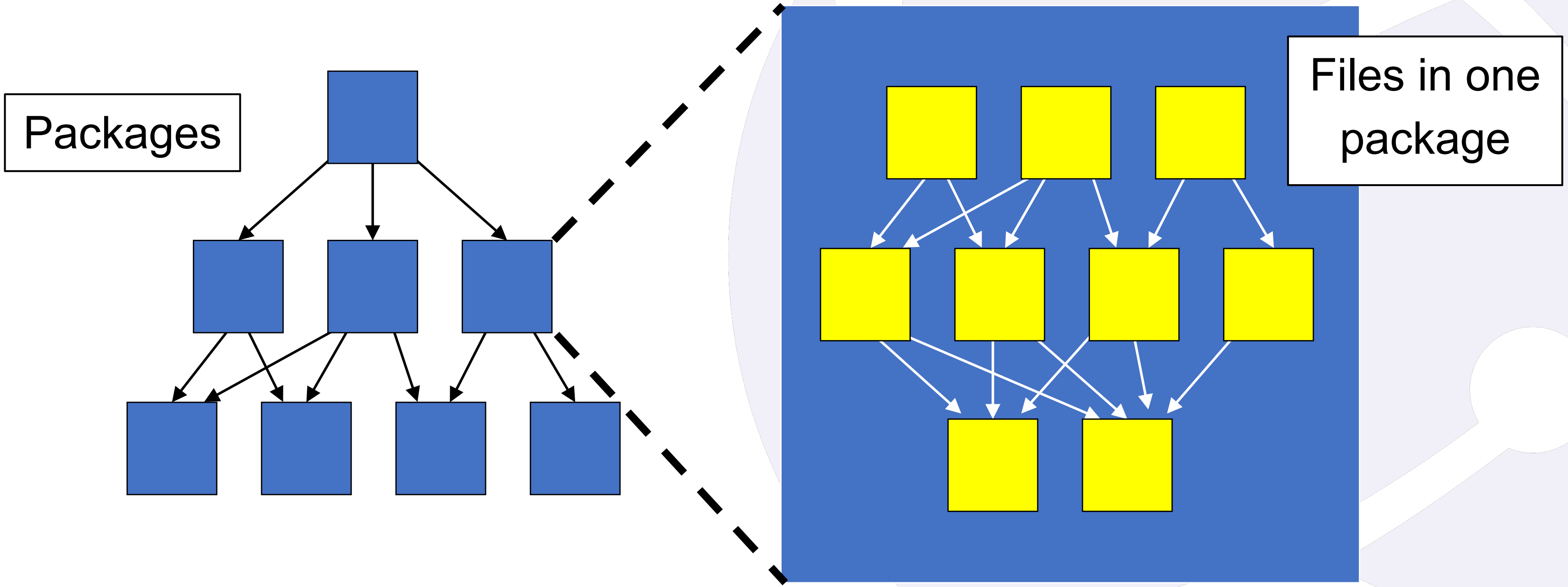
Hazards of poorly-designed code

In a poorly-designed code, it is difficult to:

- Understand the code
- Maintain the existing features
- Add new features
- Bring new developers onto the code team
- Refactor the code to support GPUs or other advanced architectures

Basic principle for large codes: Hierarchical design

- How do you wrap your brain around a large software project?
- Best answer: group related parts of the code into **packages** in a hierarchy



Separation of concerns

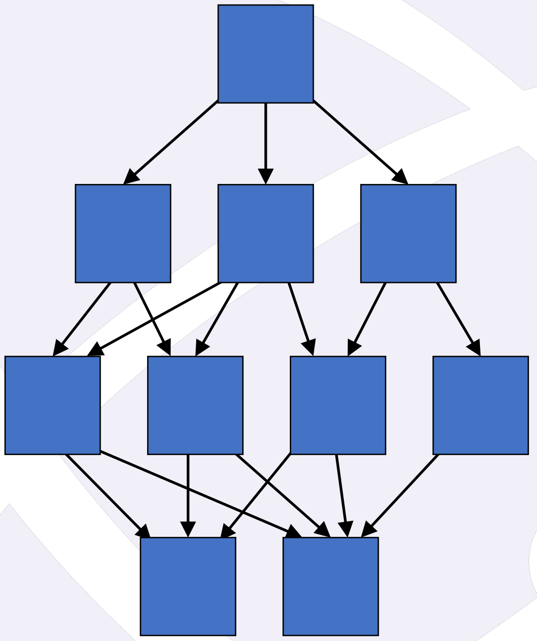
- Each package should have its own, specific area of functionality
  - In general, don’t put unrelated things in one package
  - Occasional exception: “utility” package
- This makes the code easier to understand and manage
  - In many cases, fixing an issue or adding a feature will touch just one package, or a small number of packages
  - Makes it much easier to bring new team members on board!

Encapsulation

- A class or module should have a well-thought-out interface, through which callers can interact with it
- The same holds, on a larger scale, for packages or libraries
  - Design an **application programming interface (API)** for each package
  - As long as the interface stays the same, you can modify or extend the package implementation, without have to change calling packages
  - Other developers can treat your package as a “black box” and not have to understand its details

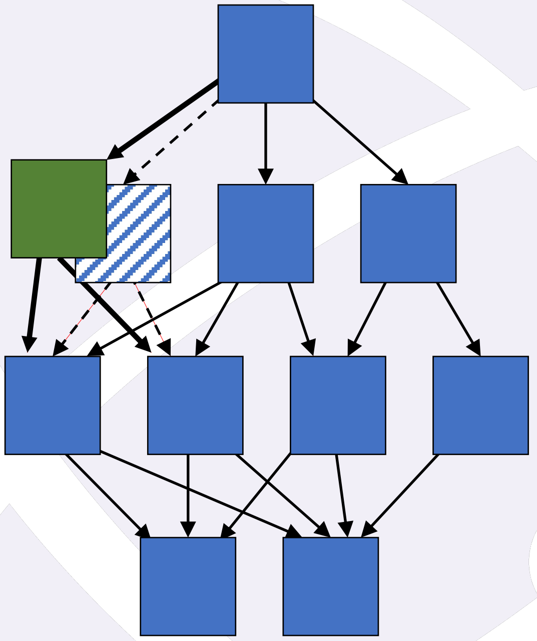
Composability

- Having a hierarchy of packages allows you to build something big out of smaller pieces



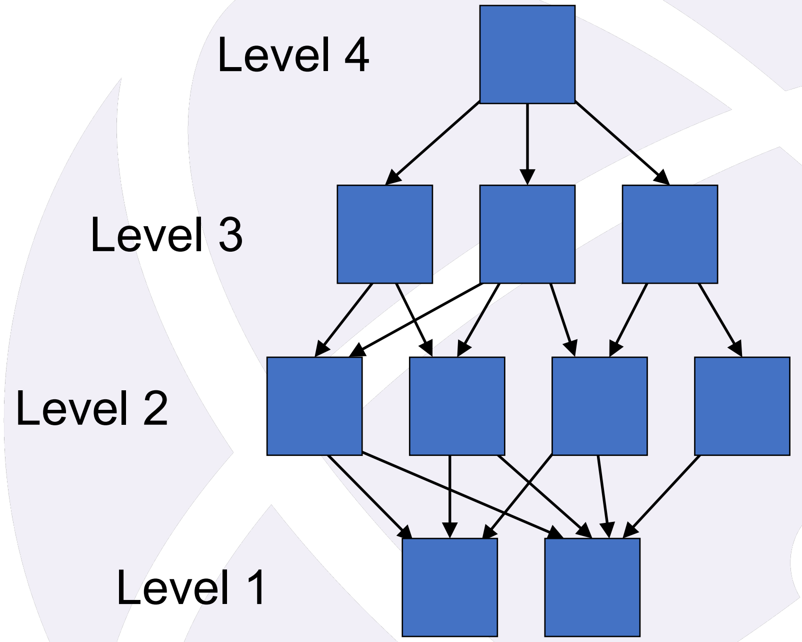
Composability

- Having a hierarchy of packages allows you to build something big out of smaller pieces
- Also allows flexibility in how each package is implemented
  - Swap out one implementation for another if API is the same
  - Put multiple implementations alongside each other, such as:
    - Multiple models with similar APIs
    - Different implementations for CPU/GPU
  - Write packages with different languages/programming models if needed



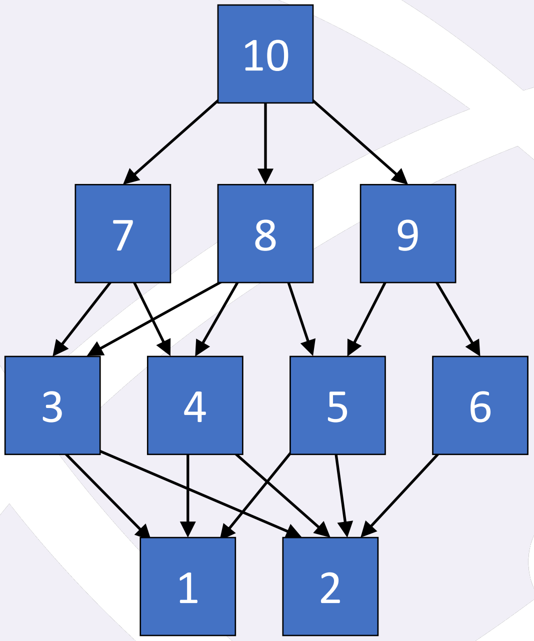
Levelization

- Can assign level numbers to a dependency graph *iff* the graph has no cycles
  - This avoids “circular dependencies”
- With levelized dependencies:
  - Always have well-defined build order
  - Can reuse a subgraph by itself if needed (e.g., new product)
  - Can test the system incrementally – start from bottom, work up
- Applies to both packages and files within a package



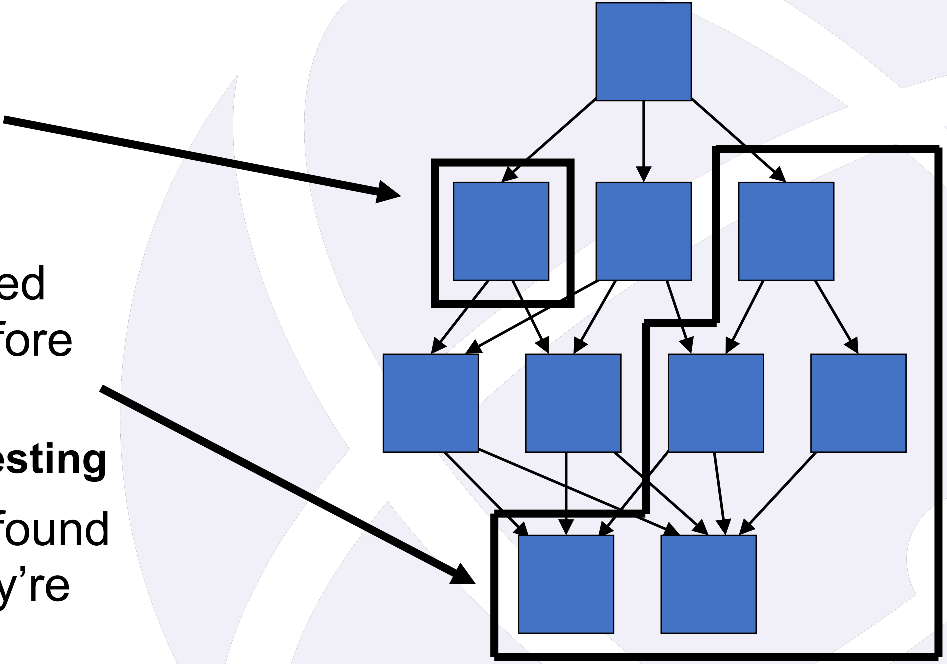
Build systems

- Packages in source tree become libraries in build system
- Can always build packages in an order that respects dependencies
- Can always construct a link line that respects all package dependencies
  - True for both the final product, and incremental tests (next slide)
  - Note: Link lines with libraries respect ordering; link lines with **.o** files do not!



Testability

- A self-contained package can be tested on its own, apart from the larger project
  - This is **unit testing**
- Subsets of packages can be tested together during development, before all packages are complete
  - This is **incremental integration testing**
- Together these allow bugs to be found earlier in development, when they’re much easier to fix



Language specifics: C/C++

- Can use C++ language features to help with encapsulation
  - Classes with inheritance, public/private data and functions, …
  - Namespaces

- Can do similar things in C, but not as much language support
  - Static data and functions to simulate “private”
  - Or, comments to mark private data and functions (“honor system”)
  - Package prefixes to simulate namespaces

Language specifics: Fortran

- Can use F90 modules to implement encapsulation
  - Declare API functions/subroutines PUBLIC
  - Declare implementation details PRIVATE
- Or, use F2003 OO features
  - Define derived types, with inheritance and type-based routines
  - Use PUBLIC and PRIVATE attributes as in C++
  - Warning: Some programming models don’t play well with this (e.g., OpenMP offload)

Style guide

- It's important to have consistency for names visible outside of a file/package
  - Class names, function names
    - **`do_the_calc()`** vs. **`DoTheCalc()`** vs. **`doTheCalc()`** …
  - File names and suffixes
    - **`#include<do_the_calc.hh>`** vs. **`#include<DoTheCalc.hpp>`** …
- It's important to have internal consistency within a single file
  - Indentation style and number of spaces
  - Naming for local variables, local functions, …
- It is nice, but not as critical, for all files to have internal elements consistent
- If you modify an existing file, follow its style! Don't impose your own
- If you contribute to an existing project, follow its style (documented or not!)

Style guide (cont’d)

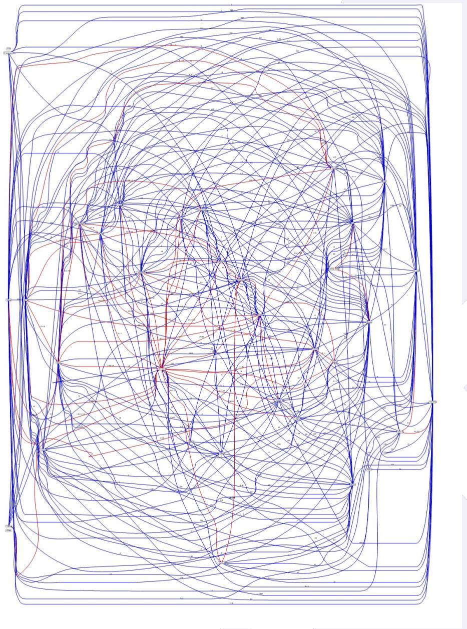
- Be careful about using someone else’s style guide
  - Their guide may have hidden assumptions that don’t apply to your project!
  - May need to do tailoring
  - Example: Google and C/C++ suffixes

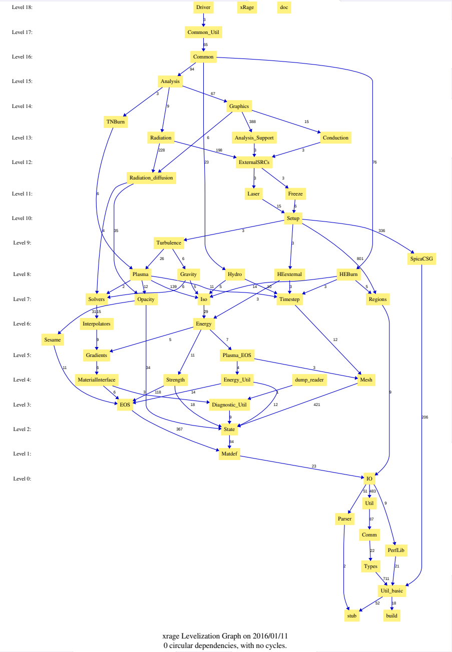
Can you improve the design of a long-running project?

Yes! (with a **lot** of effort)

- EAP project made a **major** effort in FY15-16
- Started with >20yr old code, ~470K SLOC
- In ~15 months, created ~40 packages, levelized dependencies
- Formed the basis for further code modernization work (including GPU support)

xRage dependency graphs





xrage Levelization Graph on 2016/01/11
0 circular dependencies, with no cycles.

2014-10-01

2016-01-11

Resources

- Lakos, Large-Scale C++ Software Design
  - Many of the principles apply to other languages, not just C++
  - 1996 edition: language mechanisms are way outdated, principles still apply
  - 2019 edition (volume 1 of 3): language mechanisms are hopefully more up-to-date?
- Feathers, Working Effectively with Legacy Code
  - Principles for modernizing an existing code base
- Ferenbaugh et al., Modernizing a Long-Lived Production Physics Code, SC16 poster (LA-UR-16-25446)
  - More details on the xRage refactor

Thanks for your attention!

Questions?

Los Alamos
NATIONAL LABORATORY